Multithreading for synchronization tolerance in matrix factorization

Alfredo Buttari[1], Jack Dongarra[1,2], Parry Husbands[3], Jakub Kurzak[1] and Katherine Yelick[3,4]
[1]Computer Science Department, University of Tennessee, 1122 Volunteer Blvd, Knoxville, TN, 37996, USA
[2]Computer Science and Mathematics Division, Oak Ridge National Laboratory, Oak Ridge, TN, 37831, USA
[3]Computational Research Division, Lawrence Berkeley National Laboratory, 1 Cyclotron Rd., Berkeley, CA, 94720, USA
[4]Computer Science Division, University of California at Berkeley, Soda Hall, Berkeley, CA, 94720, USA

Abstract. Physical constraints such as power, leakage and pin bandwidth are currently driving the HPC industry to produce systems with unprecedented levels of concurrency. In these parallel systems, synchronization and memory operations are becoming considerably more expensive than before. In this work we study parallel matrix factorization codes and conclude that they need to be re-engineered to avoid unnecessary (and expensive) synchronization. We propose the use of multithreading combined with intelligent schedulers and implement representative algorithms in this style. Our results indicate that this strategy can significantly outperform traditional codes.

1. Introduction

For at least two decades, HPC programmers have taken advantage of the fact that each successive generation of microprocessors would make their old software run substantially faster, either immediately or after minor adjustments. But a few main factors are converging to bring this "free ride" to an end [2]. First, system builders have encountered intractable physical barriers – too much heat, too much power consumption, and too much voltage leakage – to further increases in clock speeds. Second, physical limits on the number and bandwidth of pins on a single chip and the latency of crossing chip boundaries means that the gap between processor performance and memory performance, which was already bad, will get increasingly worse. This daunting combination of obstacles has led to unprecedented levels of parallelism as an alternative approach for continuing the increase in performance, and relatively higher cost for synchronization and communication, which we argue will require new software models and different types of parallel algorithms.

One good way to appreciate the impact and significance of this architectural change is to examine its effect on software packages that are widely familiar. The LAPACK [1]/ScaLAPACK [3] libraries fit that description. These libraries, which embody much of our work in the adaptation of block partitioned algorithms to parallel linear algebra software design, have served the HPC and

Computational Science community remarkably well for twenty years. Both LAPACK and ScaLAPACK apply the idea of blocking in a consistent way to a wide range of algorithms in linear algebra (LA), including linear systems, least square problems, singular value decomposition, eigenvalue decomposition, etc., for problems with dense and banded coefficient matrices. ScaLAPACK also addresses the much harder problem of implementing these routines on distributed memory architectures, yet it manages to keep close correspondence to LAPACK in the way the code is structured or organized. The design of these packages has had a major impact on how mathematical software has been written and used during that time. However, when one looks at how these foundational libraries can be expected to fare on large-scale multi-core systems, it becomes clear that we are on the verge of a transformation in software design at least as potent as the change engendered a decade ago by message passing architectures, when the community had to rethink and rewrite many of its algorithms, libraries, and applications.

The standard approach to parallelization of numerical linear algebra algorithms for both shared and distributed memory systems, utilized by the LAPACK/ScaLAPACK libraries, is to rely on a parallel implementation of the BLAS (Basic Linear Algebra Subroutines) - threaded BLAS for shared memory systems and PBLAS (Parallel BLAS) for distributed memory systems. Historically, this approach made tractable the job of writing hundreds of routines in a consistent and accessible manner. While this approach solves numerous complexity problems, it also enforces a very rigid and inflexible software structure, where, *at the level of LA, the algorithms are expressed in a serial way*. This obviously inhibits the opportunity to exploit inherently parallel algorithms at a finer granularity. This is shown by the fact that the traditional method is successful mainly in extracting parallelism from Level 3 BLAS (mostly matrix-matrix multiplication). In the case of most of the Level 1 and 2 BLAS, however, it usually fails to achieve speedups and often results in slowdowns. It relies on the fact that, for large enough problems, the $O(n^3)$ cost of Level 3 BLAS dominates the computation and renders the remaining operations negligible. The problem with encapsulating parallelization in the BLAS/PBLAS in this way is that it requires a heavy synchronization model on a shared memory system and a heavily synchronous and blocking form of collective communication on distributed memory systems with message passing using MPI [8]. This paradigm will break down on next generation architectures, because it over-synchronizes the code and requires large problems to amortize the synchronization overhead.

In this paper we describe our work that aims to address this situation by replacing the bulk-synchronous parallelism model with a large grain data flow model. We describe two different implementations of dense factorization routines, one for multi-core and one for distributed memory, that exploit dynamic and adaptive out-of-order execution patterns. They use higher level parallelism with a data flow execution model that avoids global synchronization, and in the cluster case use non-blocking one-sided communication from UPC [9] to fetch remote data.. Our preliminary experiments show that our implementations can yield great improvements in performance, especially on smaller problem sizes.

2. Dynamically exploiting parallelism on multi-core processors

We used the foregoing analysis of the problems of LAPACK/ScaLAPACK on multi-core systems as the basis of some preliminary tests of techniques for performing fast and efficient LA on multi-core processors. LA operations are usually performed as a sequence of smaller tasks; it is possible to represent the execution flow of an algorithm as a Directed Acyclic Graph (DAG) where the nodes represent the sub-tasks and the edges represent the dependencies among them. Whatever the execution order of the sub-tasks is, the result will be correct as long as these dependencies are not violated. This concept has been used in the past to define “look-ahead” techniques that have been extensively applied to the LU factorization. Such methods can be used to remedy the problem of synchronizations introduced by non-parallelizable tasks by overlapping their execution with the execution of more efficient ones. Although the traditional technique of look-ahead usually provides only a static definition of the execution flow that is hardwired in the source code, the idea of out-of-order execution

it embodies can be extended to broader range of cases, where the execution flow is determined at run time in a fully dynamic fashion. With this dynamic approach, the subtasks that contribute to the result of the operation can be scheduled dynamically depending on the availability of resources and on the constraints defined by the dependencies among them (i.e., edges in the DAG).

Our recent work [7] shows how the one-sided factorizations, LU, QR and Cholesky can benefit from the application of this technique. Block formulations of these three factorizations, as well as many other one-sided transformations, follow a common scheme. In a single step of each algorithm, first operations are applied to a single block of rows or columns, referred to as the panel, then the result is applied to the remaining portion of the matrix, usually called the trailing sub-matrix. The panel operations are usually implemented with Level 1 and 2 BLAS and, in most cases, achieve the best performance when executed on a single processor or a small subset of all the processors used for the factorization.

It is well known that matrix factorizations have left-looking and right-looking formulations depending on whether updates are pushed to or pulled by panels of the trailing sub-matrix. The transition between the two can be done by automatic code transformations, although this requires more powerful methods than simple dependency analysis. In particular, the technique of look-ahead can be used to significantly improve the performance of matrix factorizations by performing panel factorizations in parallel with the update to the trailing sub-matrix from a previous step of the algorithm. The look-ahead can be of arbitrary depth, as was shown, for example, in the high performance LINPACK benchmark (HPL) [4]. The look-ahead simply alters the order of operations in the factorization. A great number of permutations are legal, as long as algorithmic dependencies are not violated. From this point of view, right-looking and left-looking formulations of a matrix factorization are on two opposite ends of a wide spectrum of possible execution paths, with the look-ahead providing a transition between them. If the straight right-looking formulation is regarded as one with the look-ahead of zero, then the left-looking formulation is equivalent to the right looking formulation with the maximum possible look-ahead for a given problem.

Applying the idea of dynamic execution flow to LU factorization leads to the implementation of the left-looking variant of the algorithm, where the panel factorizations are performed as soon as possible, with the modification that if the panel factorization introduces a stall, then an update to a block of columns (or rows) of the right submatrix is performed instead. The updating continues

```
while(1) {
    fetch_task();

    switch(task.type) {

        case PANEL:
            dgetf2();    dsyrk();    dgeqr2();
                         dpotf2();   dlarft();

            update_progress();
            break;

        case COLUMN :
            dlaswp();    dgemm();    dlarfb();
            dtrsm();     dtrsm();
            dgemm();

            update_progress();
            break;

        case END:
            for ()
                dlaswp();

            return;
        }
}
```

Figure 1. Pseudo-code for the execution flow of the 3 one-sided transformations LU, Cholesky and QR.

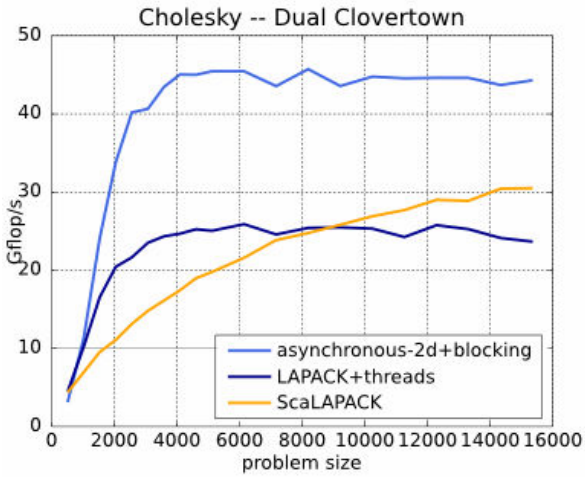

Figure 2. Comparison of parallelization techniques for Cholesky (Dual 4 core Intel Clovertown)

only until next panel factorization is possible. Figure 1 (above) shows the simplified code that defines the execution flow. Here the steps of checking dependencies and making a transition are merged into the step of fetching the next task (the fetch_task() subroutine), where the choice of transition is made dynamically at run-time depending on the progress of the execution.
Experimental results show how the dynamic workflow technique is capable of improving the overall performance while providing an extremely high level of portability. Figure 2 shows that by applying dynamic task scheduling to the Cholesky factorization, it is possible to out perform both a standard LAPACK implementation with threaded BLAS and ScaLAPACK on a multi-core processor.

3. The Distributed Memory Case

The same general philosophy also applies to the distributed memory case: tasks are executed by the processors while algorithm dependencies are respected. Some additional complications arise, however. Control is distributed in the sense that we have no central task queue and so remote task creation is required. In addition, we must deal with the entangled issues of locality, load balance, and communication latency tolerance.

In order to explore these ideas we wrote an HPL implementation from the ground up [6] using the UPC partitioned global address space language. Central to the code is a co-operative multithreading facility that allows us to take advantage of both algorithmic and communication overlap. As with the multi-core code, threads run all the major operations and dependencies are enforced by a scheduler. Our scheduler also implements task prioritization as in the multi-core case. In the distributed memory version, multiple threads share a core and they yield control of the CPU on long latency communication operations. In order to get good locality and load balance, the standard 2-D block-cyclic decomposition is used.

Dynamic look-ahead presents an interesting problem not seen in the single node case. The pieces of the matrix that participate in updates may be remote to the processor performing the computation. They must, therefore, be buffered and this uses up memory. At startup time we pre-allocate a pool of memory that will be used for these buffers. When we receive notification that a sub-task is ready to proceed, we allocate memory from the pool, transfer the remote operands (potentially yielding the CPU to mask the latency), then perform the operation. There is, however, the potential for deadlock. If the notification for a higher panel is received before that for a lower panel (perhaps due to network effects) and the higher panel grabs the last bit of memory, progress is stalled. This is avoided by always allocating memory in the order of the panel number that spawned the update with no “holes” in the sequence. In the previous example, no memory would be allocated for the higher panel (its execution is suspended) until the lower panel is handled. This strategy ensures that all updates arising from a particular panel can be buffered and so complete before dealing with any higher panels. While application specific, this suggests a possible general solution that looks at dependency information before making memory management and scheduling decisions.

Figure 3 summarizes the performance of our distributed memory code on various machines. The HPL/MPI numbers are taken from the HPC Challenge web page [5] and the UPC numbers are collected on machines that are as similar as possible (without giving any advantage to the UPC code). In all, our code is competitive with HPL (which performs static look-ahead) and outperforms ScaLAPACK (no look-ahead and synchronous) by a wide margin. In addition, the code also runs well on a single core (with all the thread creation and scheduling). It achieves 91.8% of peak on a single 1.5 GHz Itanium 2 processor and 81.9% of peak on a 2.2 GHz Opteron chip. This validates our threading strategy by demonstrating that the overhead of managing the threads is not significant.

4. Conclusion

In this report we used a case study of matrix factorizations to analyze a new software model for future HPC systems. The model relies on dynamic, dataflow-driven execution models and avoids both global synchronization and the implicit point-to-point synchronization of send/receive style message passing. In our view, highly asynchronous codes are a good fit for the massive amount of concurrency present

in these machines. Our prototype codes successfully managed to hide algorithmic and communication latencies and so deliver high performance. They are especially advantageous on smaller problem sizes and larger degree parallelism, because they avoid some of the overheads of the traditional bulk-synchronous models. We intend to further explore this programming paradigm for two-sided linear algebra algorithms (e.g., eigenvalue problems) and sparse matrix algorithms, where scalability is even more challenging and the avoidance of synchronization costs should have an even higher payoff.

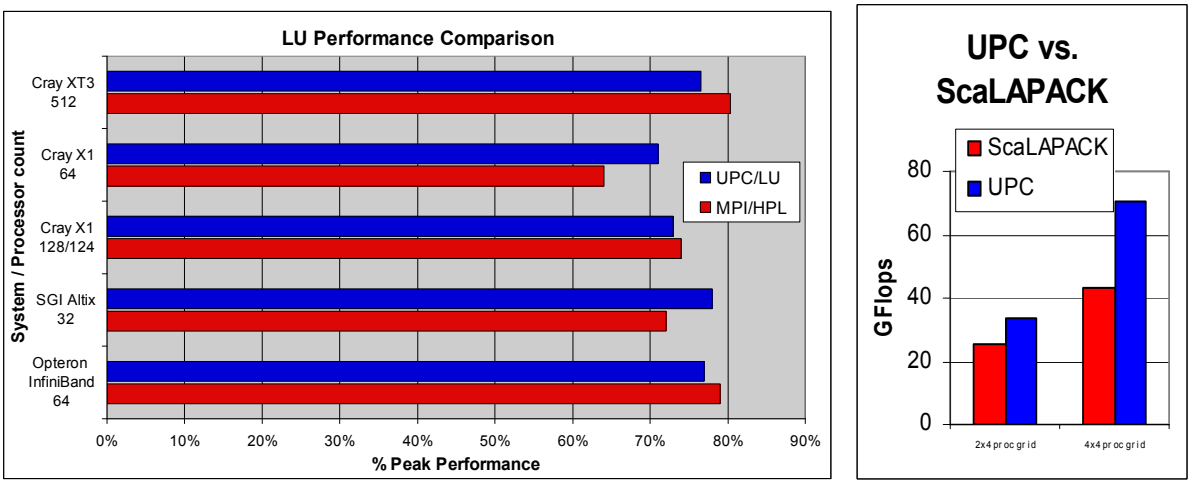

Figure 3. LU Performance Summary. The 512p HPL Cray XT3 number was estimated. The comparison with ScaLAPACK was performed on an SGI Altix with matrix sizes of 25,600 (left) and 32,000 (right).

5. References

[1] Anderson E, Bai Z, Bischof C, Blackford S, Demmel J, Dongarra J J, Du Croz J, Greenbaum A, Hammarling S, McKenney A and Sorenson D 1999 *LAPACK Users' Guide – Third Edition* (Philadelphia: SIAM Press)

[2] Asanovic K, Bodik R, Catazaro B, Gebis J, Husbands P, Keutzer K, Patterson D, Plishker W, Shalf J, Williams S and Yelick K 2006 The landscape of parallel computing research: a view from Berkeley *UC Berkeley EECS Technical Report UCB/EECS-2006-183*

[3] Choi J, Dongarra J J, Ostrouchov S, Petitet A, Walker D and Whaley R C 1996 The design and implementation of the ScaLAPACK LU, QR, and Cholesky factorization routines *Scientific Programming* **5** pp 173-84

[4] Dongarra J J, Luszczek P and Petitet A 2003 The LINPACK benchmark: past, present, and future *Concurrency and Computation: Practice and Experience* **15** 9 pp 803-20

[5] HPC Challenge Benchmark Page 2007 http://icl.cs.utk.edu/hpcc/

[6] Husbands P and Yelick K 2007 Multi-threading and one-sided communication in parallel LU factorization *To Appear in Proc. SC'07(Reno, NV, USA, November 2007)*

[7] Kurzak J and Dongarra J J 2006 Pipelined shared memory implementation of linear algebra routines with lookahead - LU, Cholesky, QR Proc. *Workshop on State-of-the-Art in Scientific and Parallel Computing (Umeå, Sweden, August 2006)*

[8] Snir M, Otto S, Huss-Lederman S, Walker D, and Dongarra J J 1998 *MPI: The Complete Reference - 2nd Edition: Volume 1*. (Cambridge: MIT Press)

[9] UPC Consortium. UPC Language Specifications 2007 http://upc.lbl.gov/docs/user/upc_spec_1.2.pdf